A New Generation of Systematic Programming Tools

James R. Larus
larus@microsoft.com
Microsoft Research
One Microsoft Way
Redmond, WA 98040
http://research.microsoft.com/~larus
http://research.microsoft.com/spt/

Improving software and software development requires a new generation of programming languages and tools that make it possible to apply the enormous computational resources on a programmer's desk to the problem of finding errors and inconsistencies in programs. Although these tools alone will not find, let alone eliminate all programming errors, they have already demonstrated that they can improve program quality and reduce development cost. The Software Productivity Tools group in Microsoft Research has developed a variety of programming tools, which use simple, partial specifications and sophisticated program analysis to find errors systematically. This paper explains why this approach is beneficial, describes some existing tools, and points out the many open research directions.

1 Introduction

This paper briefly outlines the research of the Software Productivity Tools (SPT) group in Microsoft Research. This group, in collaboration with an internal tools group (Programmer Productivity Research Center), is pursuing a focused strategy to improve both the quality of software and the effectiveness of developers. Our approach combines better and more expressive programming languages with stronger, semantically aware programming tools. The goal is to develop a new programming style that exploits the enormous computational power of today's computers to detect systematically errors and inconsistencies in large software systems. This research began when the author spent his sabbatical at Microsoft and observed that programmers there, like programmers elsewhere, relied on tools conceived before the programmers themselves. Computers and the scale of programming have changed dramatically over the past 20–30 years, but our primary tools—compilers, editors, debuggers, source code control systems, bug databases, etc.—are lineal descendents of tools developed in the heyday of timesharing.

These time-honored tools deal with the mechanics of programming, but provide little help in detecting most programming errors, which still are found by testing, an expensive and ultimately incomplete method. It is accepted wisdom that the earlier in a development cycle that an error is found, the less it costs to correct it. Nevertheless, in today's practice, it is not uncommon for a simple programming error, such as a leaked resource, to lurk for months or years. Finding and fixing the bug may involve several people beyond the developer who introduced the error, such as a customer support technician to find out about the error from customers, a tester to reproduce the error, and a (perhaps different) developer to diagnose the error and correct it.

Most tools provide little assistance in finding bugs because of their superficial understanding of language semantics and complete ignorance of the abstractions used to construct a program. Without mechanical assistance, quality programs will remain difficult and time consuming to produce. Our belief is that in programming, as in most human endeavors, appropriate tools can simultaneously improve both quality and productivity. Tools that quickly and systematically detect errors and fully explain their causes will lead to better software and reduce programmer (and tester) effort. These tools are not a panacea, as innovative approaches to program design, comprehension, structure, development, and testing are also badly needed.

Briefly, SPT's approach has been to elicit supplemental information from programmers and to use these partial specifications to detect inconsistencies and errors in the actual code. The closest analogy to this work is type systems in programming languages. Although some languages—for example, Lisp and, more recently, scripting languages such as tcl or perl—do not have explicit types, programmers have come to recognize the compelling advantages of rich type systems, such as those in ML or Java. Types enhance documentation of programmer intent, provide a foundation for constructing abstractions, statically detect certain errors, and facilitate compiler optimization. Moreover, types give lie to the claim that programmers will not write specifications, as types are specifications of a limited aspect of a program's behavior. Programmers happily write type declarations because they are generally easy to produce and understand, improve the comprehensibility of code, and detect a significant number of errors. These attributes are essential in all tools that demand programmer effort.

Another, complementary way of looking at this research is that it is applying the classic engineering principle of redundancy to build reliable systems from unreliable components. The redundancy in this case comes from a non-executable specification, which is not needed to specify the algorithm in a program, but instead serves the twin roles of increasing programmer comprehension and exposing inconsistencies that may indicate errors. The latter task has been the focus of our research, though the former may prove more significant in the long run.

This paper briefly describes several tools developed by the SPT group and other researchers, and concludes by looking into the future.

2 SPT Research

The SPT group has built several tools that use partial behavioral specifications to find errors in programs. A behavioral specification constrains a program's temporal sequence of actions, as contrasted to traditional types, which specify legal operations on values. For example, consider the Unix socket API, which manipulates `sockaddr`'s and `sockets`. Type checking can ensure that an appropriate value is passed to each routine, but it cannot check that an object created in a call on `socket` is bound by a call on `bind` before it is passed to the `connect` function. Many, but certainly not all, programming errors arise from disobeying a rule of this sort along one or more paths through a program. To compound the situation, these rules are rarely stated precisely or completely, but instead are described by a combination of prose and sample code.

The first three projects described below—SLAM, ESP, and Vault—explore different approaches to specifying program behavior at interface boundaries and utilize different techniques to validate that code follows its specification. The fourth project, Behave!, extends this approach to concurrent systems.

2.1 SLAM

SLAM checks temporal safety properties of C programs. A programmer writes these properties in a language called SLIC and then feeds them and a program to the SLAM tool, which applies a model-checking-like approach to verify that the code obeys its specification [1-3]. SLAM uses an abstraction called a boolean program to model a program. A boolean program is the program's skeleton, consisting only of simple control flow and boolean computation. SLAM starts with a simple program model and iteratively refines it into an increasing accurate model of a program's possible behavior. SLAM uses this model to determine whether a feasible path will violate a rule in the SLIC specification. If the current model cannot answer this question, SLAM produces a more accurate model and continues the process.

SLAM is a fully automatic tool, which does not require programmer annotation, beyond the specification of the program properties. When SLAM detects an error, it provides a detailed explanation of why the error occurs, including the complete interprocedural path and values of variables leading to the error. Moreover, unlike most tools, SLAM can avoid reporting "false

positives," that is, artifacts due to the analysis process. However, since non-trivial program analysis is undecidable in general, SLAM may fail to determine whether a property holds or not. Despite this possibility, we believe that this is an interesting point in the space of programming tools, for many programmers would value a small number of extremely precise error reports, even if a tool occasionally says "I don't know."

We have built a working version of SLAM and are currently using it to find errors in Windows device drivers, which are relatively small pieces of code that form a critical and error-prone portion of all operating systems. Moreover, the many families of existing drivers require a tool, like SLAM, in which specification is distinct from code. SLAM has successfully found previously unknown errors in drivers.

2.2 ESP

ESP is a related project that is focused on a very different point in the space of tools. It looks for a similar class of errors in very large (millions of line) C and C++ programs. ESP builds on our previous research on efficient global value flow analysis [7, 10] and takes a more traditional program-analysis approach to this problem. Like SLAM, ESP's analysis is sound, so it does not miss possible errors in a program. However, because of its scalability goals, ESP tolerates some false positives, though the hope is to keep them at an acceptable level by careful program analysis.

ESP breaks the problem of tracking behavioral properties into three simpler problems: tracking the sequence of operations along a control-flow path, tracking the flow of values between operations, and determining the feasibility of program paths. Each of these problems can be solved by a scalable program analysis technique, and these separate analyses can be combined to track SLIC-like properties through a very large program. The trade-off, of course, is precision. Each analysis is itself a conservative approximation to a program's behavior, and ESP itself introduces limitations. For example, to ensure that the analysis remains tractable, ESP tracks the state of each object separately, so it cannot detect correlations among objects' state, which might rule out infeasible program behavior that appears to be an error.

However, the tradeoffs in ESP are based on the intuition that there is a fundamental difference between the way a good programmer reasons about control and data flow in his or her own code and in code written by others. A programmer is more likely to rely on correlations among actions in her own code, but confirm actions in others' code through explicit testing. If this hypothesis is correct, ESP should be able to identify correct code (and therefore avoid many false error reports) through a combination of strong local analysis and weak global analysis. ESP has been used to verify file I/O behavior of the gcc compiler.

2.3 Vault

The Vault programming language takes a very different approach to specifying and checking program behavior [8]. Unlike SLAM and ESP, which separate a specification from a program, Vault provides a rich type system that allows a programmer to describe a function's behavior as part of its type signature. When a caller of this function is compiled, the type checker ensures the behavioral rules, as well as the usual type rules.

Vault enforces interface rules by statically tracking programmer-designated resources through keys. A resource key cannot be duplicated or thrown away. Memory, for example, can be treated as a resource to prevent errors such as dangling pointers and memory leaks. A block of memory named by a key is accessible so long as the program holds the corresponding key in scope. Another example is a key representing access to a shared object. The key comes into scope through the acquire lock operation. Since operations on the shared object require its key, the object can only be accessed while the key is in scope. Releasing the lock takes the key out of scope, and access to the shared resource is lost.

In general, resource keys enable a programmer to describe application- or API-specific rules, such as finalization, operation ordering, life-time dependencies between different data objects, and mutual exclusion in multithreaded environments. A function signature can specify which resource keys must be held, and their state, before a call on the function and which keys will be held on return. A compiler can check these rules each time that code using resource-annotated types is compiled.

Vault offers some compelling advantages because a specification is integrated into a program, where it increases programmers' understanding of the code. Moreover, the consistency of the code and specification is checked each time the program is compiled. However, to ensure that type checking remains fast and efficient, Vault imposes some restrictions—for example, that the state of an object must be identical along all paths that meet at a point in the program—that are not necessary in other approach, such as SLAM, which tracks values along individual paths through a program. We used Vault to write a floppy disk driver for Windows 2000 and to capture some of the rules for Microsoft's DirectX library.

2.4 Behave!

The Behave! project is focused on specifying and checking the behavioral properties of asynchronous programs, which are a particularly challenging and error-prone type of program [5]. This programming style is common in operating system kernels; distributed systems; and other event-driven, message-passing programs. Because of their asynchrony and concurrency, these programs are difficult to write, understand, or modify and are nearly impossible to test thoroughly. This project uses a behavioral type system to specify the intended behavior of an asynchronous program and then uses model-checking techniques to verify that the code obeys its specification.

3 Related Projects

This line of research has attracted considerable interest recently (most of which is not described here), and several tools have come into effective use. Intrinsia developed Prefix [4], which, although it performs an unsound, incomplete program analysis, has nevertheless proven effective at Microsoft in finding many common programming errors, such as null-pointer dereferences and misuse of common APIs. Prefix analyzes a limited, but carefully selected, set of program paths. Along each path, it examines the effects of each statement, to ensure its preconditions—such as a pointer being non-null when dereferenced. Because Prefix's analysis is heuristic, the tool pays considerable attention to explaining errors and presenting them in an order likely to favor real errors over false positives. In practice, this aspect of the tool is crucial to programmer acceptance, particularly for large programs that produce thousands of warnings.

More recently, Engler built a tool called Metal [6, 9], which uses modified version of gcc to find errors in the Linux kernel. Metal starts with a programmer-supplied specification of a programming convention or interface behavior, which is written as a finite-state automaton. Metal examines each function in a program by tracking the automaton along all program paths and reporting transitions into error states. Interprocedural information is provided by an ad-hoc prepass over the program, which captures and records properties of functions for use in the local analysis phase. Engler's approach has demonstrated that even limited program analysis can be effective in finding thousands of errors in the Linux kernel. More recently, he has used Metal to infer program specifications, by looking for sequences of actions that occur predominately in one order, under the assumption that the rare exception to a rule is worth examining.

4 Discussion and Future Work

These are among the first of a new generation of programming tools that have the potential to improve both software quality and programmer productivity. Realistically, no one contends that these tools are Brook's "silver bullet" that eradicates programming problems. Even with greatly improved tools, programming will remain a challenge and some errors will be found only when a program runs.

This is to be expected, as these tools deliberately narrow their focus to realistic, attainable goals such as partial specification and error detection. The older, more ambitious approach sought total correctness—the absence of errors—which proved impossible in practice and produced no tangible benefits for most programmers.

Experience shows that limited specification is far better than no specification, both because partial specification opens the possibility of automatically validating program properties, and also because precise, concrete specifications reduce programmer misunderstanding, which is the ultimate source of many errors. Programmers rarely write precise documentation or specifications because non-executable text of this sort takes time and provides few benefits. However, when a specification becomes an input to a tool, its value to a programmer increases, as does the incentive to produce and maintain this type of documentation. Moreover, tools enhance the credibility of this type of specification, since their currency and accuracy is guaranteed by the verification that connect them to the actual code.

Nevertheless, there is need for considerably more research and development in this area. We have only recently moved beyond types and values to start exploring which other aspects of a program can be stated and checked automatically. Much of the current research focuses on temporal sequences of actions, but we do not yet have the experience to know if current specification languages are expressive enough or whether they and the analysis tools detect important classes of errors. Moreover, these properties are very low-level and often difficult to relate to important issues such as performance, reliability, or security. Concurrency poses hard problems that are becoming increasingly urgent as programming moves into the distributed world made possible by the Internet. In addition, the program analysis that underlies these tools needs further improvement. Although great strides have been made in recent years in interprocedural and pointer analysis and software model checking, the techniques too often offer a stark choice between scalable or precise algorithms.

5 References

[1] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani, "Automatic Predicate Abstraction of C Programs," in Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI01). Snowbird, UT, June 2001, pp. 203-213.

[2] Thomas Ball and Sriram K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," in Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001). Toronto, Canada, May 2001, pp. 103-122.

[3] Thomas Ball and Sriram K. Rajamani, "The SLAM Toolkit," in Proceedings of the 13th Conference on Computer Aided Verification (CAV'01). Paris, France, July 2001.

[4] William R. Bush, Jonathan D. Pincus, and David J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," Software-Practice and Experience, vol. 30, num. 5, pp. 775-802, 2000.

[5] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof, "Types as Models: Model Checking Message-Passing Programs," in To appear: Proceedings of the Twenty-Ninth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02). Portland, OR, January 2002.

[6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler, "An Empirical Study of Operating Systems Errors," in Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP18). Alberta, Canada, October 2001.

[7] Manuvir Das, "Unification-based Pointer Analysis with Directional Assignments," in Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI 00). Vancouver, BC, June 2000, pp. 35-46.

[8] Robert DeLine and Manuel Fähndrich, "Enforcing High-Level Protocols in Low-Level Software," in Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01). Snowbird, UT, June 2001, pp. 59-69.

[9] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," in Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP18). Alberta, Canada, October 2001.

[10] Manuel Fahndrich, Jakob Rehof, and Manuvir Das, "Scalable Context-Sensitive Flow Analysis Using Instantiation Constraints," in Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI 00). Vancouver, BC, June 2000, pp. 253-263.